BUILDING SIMULATION MODELS WITH MODSIM: A CASE FOR OBJECT-ORIENTATION

A Thesis Submitted

in partial Fulfilment of the Requirements

for the Degree of

MASTER OF TECHNOLOGY

by

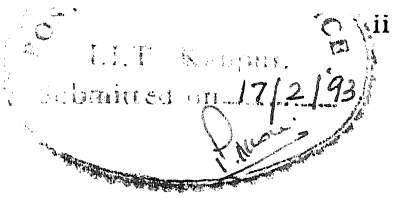
V.K.PRABHAKAR

to the

DEPARTMENT OF INDUSTRIAL & MANAGEMENT ENGINEERING

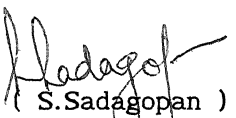
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

FEBRUARY 1993



CERTIFICATE

This is to certify that the present work entitled "*Building Simulation Models with MODSIM: A Case For Object-Orientation*", by *V.K.Prabhakar*, has been carried out under my supervision and has not been submitted elsewhere for the award of a degree.



(S.Sadagopan)
Professor
Industrial & Mgmt. Engineering
Indian Institute of Technology
Kanpur - 208 016

February, 1993

ABSTRACT

Simulation is one of the most widely used techniques in managerial decision making. In the process of developing a validated, useful simulation model, the modeler must make a number of methodological decisions, one of which is the selection of a computer language for implementing the model. Traditionally, the majority of simulations have been implemented through the use of general purpose programming language or through special purpose simulation languages. However, these languages have drawbacks and limitations and fall below the expectations of the modellers.

Object-Oriented programming (OOP) is a new method for designing and implementing software systems. It has powerful features like data abstraction, encapsulation, inheritance and polymorphism, all of which significantly change the practice of programming. MODSIM is a general purpose, modular, block-structured high level programming language which provides direct support for object-oriented programming and discrete-event simulation.

The present work is directed towards demonstrating how the power of object-orientation can improve modelling and implementation of simulation models. Some reasonably complex models are designed and implemented using MODSIM. The approach is compared with traditional approach and superiority of the object-orientation is explained.

ACKNOWLEDGEMENT

I express my profound gratitude and indebtedness to Dr.S.Sadagopan for his invaluable guidance, both in the thesis and out of it. I consider myself to be most fortunate to have worked under him and I cannot forget this finest experience of meeting Dr.Sadagopan in my life time. I also express heart-felt thanks to him for exposing me to the newest concept of software - OOP.

I am grateful to Mrs.Sadagopan for her care during my stay at IIT, Kanpur. Special thanks for innumerable coffees and snacks.

I thank Jones, Reddy, Sridhar, Sudhakar and other friends for making my life more enjoyable. I also thank Sreekant for his company during course work.

I would like to take this opportunity to thank all my teachers. Due thanks to *IME family*.

This acknowledgement would be incomplete without thanking the person who brought cheer into my life at IIT, Kanpur - Giridhar! Thanks.

V.K.Prabhakar

TO MY PARENTS

FOR THEIR LOVE & SUPPORT

CONTENTS

LIST OF FIGURES

CHAPTER I
INTRODUCTION

1.1 INTRODUCTION

Simulation is one of the most widely used techniques in managerial decision making. Recent surveys on the use of operations research methodologies place simulation analysis either first or second to linear programming applications [14]. This can be attributed to the advent of computer-based simulation, software availability, and the technical developments of simulation methodology. Moreover, this popularity has in turn generated a demand for better approaches for developing and implementing simulation models, especially when systems being modelled are large and/or complex.

In the process of developing a validated, useful simulation model, the modeler must make a number of methodological decisions. One important decision is the selection of a computer language for implementing the model. Traditionally, the majority of simulations have been implemented through the use of general purpose programming language such as FORTRAN, Pascal, C, and BASIC. Besides, there are over one hundred special purpose simulation languages available, like GASP, GPSS, SIMAN, SIMSCRIPT and SLAM. Unfortunately, these languages do have some drawbacks and limitations and fall below the expectations of the modellers. The basic cause for this short-fall is that these languages are based on what is known as *procedural paradigm*. They all share a common approach to solving a problem, differing only in their syntax [9].

A *programming paradigm* provides the system designer with techniques that guide problem solution. Recently, there has been growing interest in utilizing some of the alternatives to the procedural paradigm to facilitate the solution of certain type of problems. For

example, the rule-based paradigm has been widely used in the development of expert systems. The access-oriented paradigm has proven to be useful approach for building user interfaces. Similarly, *object-oriented paradigm* is viewed as appropriate for modelling and implementing simulation models [1, 19].

1.2 OBJECT-ORIENTED PROGRAMMING

Object-oriented programming (OOP) is a relatively new method for designing and implementing software systems. Its major goals are to improve programmer productivity by increasing software *extensibility* and *reusability* and to control the complexity and cost of software maintenance.

Building a large program with traditional approach typically involves a number of different data structures, functions, and other elements. Building the program, adding capabilities and eradicating bugs is a constant process of changes, among many programmers at the same time. Even a small change in one part of the program can have unforeseen consequences elsewhere. Fortunately, OOP gives the solution for this! [6]. *It enables a program to be written with a focus on the description of the problem rather than algorithms for solving the problem.* It usually requires less coding and is easier to modify. The focus of OOP is on the *data* to be manipulated rather than on the procedures that do the manipulating. That is, the data form the basis for the software decomposition.

The most common definition of OOP centers around several major concepts: data abstraction, encapsulation, inheritance, polymorphism, and dynamic binding.

1.2.1 DATA ABSTRACTION

Abstract data types are the centerpieces of OOP. An abstract data type is a model that encompasses a type and an associated set of

operations. These operations are defined for and characterize the behaviour of the underlying type.

A *class definition* describes the behaviour of the underlying abstract data type by defining the interface to all the operations that can be performed on the underlying type. The operations themselves are known as *methods*.

With the above definition of abstract data type and class, an *object* can now be defined as a variable declared to be of a specific class. In other words, object represents an instance of the class. Hence, they are dynamically allocated data structures coupled with methods. Such an object *encapsulates* state by containing a copy of all the fields of data and methods that are defined in the class definition. Actions may be performed on such an object by invoking one or more methods defined in the class definition. The process of invoking a method is called sending a *message* to the object. Such a message typically contains parameters just as in a procedure or function call invocation in a non - object-oriented language. The invocation of a method typically modifies the data stored in the particular object. If several objects are defined to be of the same class, they will typically contain set of values different from each other.

1.2.2 ENCAPSULATION

Encapsulation is the process by which individual software objects are defined. It defines:

1. A clear boundary that encompasses the scope of all the object's internal software.
2. An interface that describes how the object interacts with the other objects.
3. A protected internal implementation that gives the details of the functionality provided by the software object. The implementation

details are not accessible outside the scope of the class that defines the object.

1.2.3 INHERITANCE

Each object of a class contains its own set of values of the underlying data and can manipulate this data by receiving messages. The usefulness of such data encapsulation is apparent when another user can customize the class for another application by adding to or modifying methods from the given class and use it. In other words, a new object type has been defined in terms of the existing object type. The feature of the object-oriented language that makes this possible is called *inheritance*. The newly derived class is termed *derived class* or *subclass* of the *parent class* or *base type*. Inheritance allows programmers to reuse all or parts of an existing class in constructing a hierarchy of reusable software components.

The derived class will normally include new fields and/or methods not present in the parent class. It may also redefine (override) the implementation of a method defined in an underlying object type.

1.2.4 POLYMORPHISM

In object-oriented languages, if class P is a parent of subclass S, then an object s, of subclass S, can be used wherever an object p, of parent class P, can be used. This implies that a common set of messages can be sent to objects of class P and class S. Whenever the same message can be sent to objects of a parent class and objects of its subclass, this is defined as polymorphism.

The primary advantage of polymorphism is in program testing [16]. Once the base class is written and debugged, its code is available to the derived classes. The code in the derived class is then implicated during testing when a program bug is found. Knowing where a bug occurs is a

big time-saver during testing.

1.2.5 DYNAMIC BINDING

Dynamic binding is the process in which all linking occurs at program execution time; therefore all objects are defined at run time, and their functions depend on the circumstances at the time of program execution [6].

In strongly typed languages, the type of each operand is fixed at compile time, thus statically determining the operation that will be performed when the program is run. An object-oriented language requires the determination of an action to be deferred until the program is run, because of dynamic binding.

1.3 SIMULATION PROGRAMMING & OBJECT-ORIENTATION

At the most basic level, simulation programming is no different from any other kind of programming, and all the standard techniques of software engineering for ensuring good design can and should be used. To quote Hoare (1981):

> "...there are two ways of constructing a software design: One way is to make it so simple that there are *obviously* no deficiencies and the other way is to make it so complicated that there are no *obvious* deficiencies."

The point that, selection of a computer language for implementing the simulation model is a crucial decision the modeler has to take, need not be over-emphasized. It has great impact on achieving useful results from the model. And the modeler is left with a host of general purpose and special purpose simulation languages that have drawbacks and limitations in many aspects.

Bratley *et. al.* [2] discuss at length these limitations and have even suggested what a modern simulation language *ought* to look like. Although it is not hard to write simulation in a general purpose language, it may be tedious.This has its own dangers also: it is easier to overlook

mistakes in a long program than in a short one. Worse, things are more complicated when the system being modelled is large and/or complex. On the other hand, one is not comfortable writing in a language whose behaviour one is not able to understand and predict in detail which may lead to subtle errors and misconceptions. Also, maintenance of such program may also prove troublesome. Hence the necessity to define and implement a modern simulation programming language with adequate simulation capabilities and with following properties:

First, like any other modern language, a simulation language needs an adequate formal syntactic structure, for efficient compilation and programming ease. Of the present languages, except for SIMULA, no other simulation language meets this demand. Next, the semantics of the language should be clear and well-defined. A certain informality in definition seems necessary if clarity is not to suffer. SIMSCRIPT is muddled, SIMULA's semantics are opaque and only GPSS meets this requirement. The facilities offered by the language should include an easy-to-use mechanism for defining and manipulating powerful data structures. An efficient clock mechanism is essential. The language should include a battery of random number generators; good facilities are needed for collecting statistics and printing reports. If existing program libraries are easily accessible, so much the better. And of course, all these facilities should not be obtained at the price of catastrophic run-time efficiency.

Besides, the language should be English-like, self-documenting, and readable by the user, who is primarily interested in the system under study, not computer programming.

Finally, there are a number of simple-seeming problems, yet potential, in simulation language design that need a good solution. The classic example is the *'wait - until/for'* problem: if a language includes

some such construct as

wait until/for (Boolean expression)

the simple way to implement this is to check the value of the Boolean expression after every simulation event. However, this may be inefficient and even wrong! For example, consider a situation where customers queue for a resource. If they wait for, say, 100 units of simulation time afterwhich they leave the queue and go elsewhere (this is called *reneging*). In that case the Boolean expression is simply

time = 100

If no event is scheduled to occur at this time, the condition will never be satisfied! None of the known ways to alleviate such problem is entirely satisfactory.

This long list of requirements might have led Bratley *et. al.* to think, as they mentioned in their book, that until a paragon of languages appears, programmers have to make do with what exists. But, fortunately, the object-oriented paradigm has responded in establishing a modern language with all the above mentioned features, and it is - MODSIM.

1.4 OVERVIEW OF MODSIM

The Modular Simulation language, MODSIM II, is a general-purpose, modular, block-structured high level programming language which provides direct support for object-oriented programming and discrete-event simulation. It is intended to be used for building large *process-based* discrete event simulation models through modular and object-oriented development techniques. Its syntax and control mechanisms are similar to that of Modula-2, and resembles Modula-2, Ada and Pascal.

Programs written in MODSIM are organized around object types. Objects are dynamically allocated data structures coupled with methods. The

fields and methods are collectively known as its properties. The fields in the object data structure define its state at any instant in time while its methods describe the actions which the object can perform. The following are the features of the object type, which is a large leap from simple types:

* The programmer creates new instances of any object type dynamically, as needed, and disposes them when they are no longer needed.

* Fields and methods can be private and/or public in nature.

* There can be (and usually are) multiple instances of each type of object existing concurrently in a MODSIM program.

* Only an object instance can change the value of any of its fields using its methods.

* Code in any part of the program can ask an object instance the value of any of its public field.

* An object instance's method can be invoked by sending a message to the object asking it to perform a particular method.

More powerful features come in the form of new programming capabilities added to the programmer's tool box by the objects. Although these new capabilities are more easily classified as evolutionary than revolutionary, the effect of their introduction on programming technique and style has been profound. Some of the new capabilities provided by the objects are:

* **Encapsulation of data & code :** Tying together the fields which describe the object's state with the methods which describe its behaviour. Controlling access to the fields.

* **Inheritance :** Once an object type has been defined, new types can be defined based on the existing types. Each descendant in the hierarchy can add its own fields and method definitions to those of ancestors.

* **Polymorphism :** Allowing different object types in a hierarchy to share the same method name, but provide their own definition of that method. This results in a generic invocation producing different behaviours appropriate to the object being referenced.

MODSIM has powerful and flexible capabilities for dealing with discrete-event simulation. Simulation is supported directly by built-in language constructs and by a library module which contains a number of objects and support procedures. Finally, simulation time is automatically maintained by MODSIM.

The simulation capabilities of MODSIM are discussed in detail in the following chapter.

1.5 LITERATURE SURVEY

Considering the fact that the concepts of object-oriented paradigm are of recent origin and with the absence of a programming language with support for both OOP and simulation, it can be expected that the literature does not contain any attempts to test the capabilities of a language with support for both OOP and simulation, for building simulation models. This is especially true with MODSIM, which is of very recent origin.

However, some attempts were made to demonstrate the appropriateness of other OOP languages, especially C++, for developing discrete-event simulations. Wiener and Pinson [18] have developed two types of queueing models, *viz.*, a multi-server model with a single queue and a multi-server model with separate queues. These models are simple in nature, and are built mainly to demonstrate the use of inheritance in C++ as a means of achieving easy, localized maintenance. Otherwise, the model does not explore the capabilities of OOP for simulation purposes.

Eldredge *et al.*[9] have made a serious attempt to explain the appropriateness of an object-oriented programming language for discrete-event simulation. They simulated a single-server queueing model with C++. This model also lacks the complexity of system required to prove the point about advantages of building simulation models with an object-oriented programming language.

Corey and Clymer [7] attempted to use a object-oriented programming language for simulation purpose from a different perspective. They attempted to show how computer programs modelling object movement and interaction can be converted from a FROTRAN style, fixed time step, static memory allocation simulation to a modern language, variable time step, dynamic memory allocation simulation. A set of routines were written in Pascal and an airport simulation is demonstrated. The object movement and interaction are demonstrated which are natural for an OOP language.

Zeigler [19] also reports the work on discrete-event modelling in an object-oriented environment. His concentration was mainly on developing the model with the power of inheritance and by modular development.

1.6 OVERVIEW OF THE THESIS

The main objective of the present work is to study how the power of object-orientation can improve modelling and implementation of simulation models compared to procedural languages. In other words, the goal is to study how simulation modelling and implementation in an object-oriented programming language is different from a traditional procedure-oriented language. Towards this end, MODSIM is chosen for building models with the power of object-orientation.

The systems to be modelled are designed in such a way that

they represent the majority class of systems, the kind of which are subjected to simulation analysis. They are also made reasonably complicated by adding some features of practical importance. Attempt is also made to see that the models demonstrate various object-orientation capabilities. The systems selected are queueing and inventory models, the details of which are given later.

Hence, the concentration of the work has been on building and implementing the models in MODSIM. And wherever appropriate, comparison is made with the traditional approach.

The organization of the report is as follows: The first chapter is used for introducing the concepts of OOP and the motivation for going for an object-oriented programming language for building simulation models. It also gives the overview of the object-oriented language, MODSIM, with simulation capabilities.

The second chapter is devoted for describing the techniques in traditional simulation programming and then goes into the details of MODSIM as a simulation language. It may be noted that presenting the syntax of the language is not the idea and only used wherever a point is to be made.

The third chapter is exclusively for describing the models that were selected and their implementation in traditional languages and in MODSIM, with stress on the latter. The benefits of the object-orientation modelling are explained subsequently.

The fourth chapter summarizes the work and tries to define the direction of further work.

CHAPTER II
SIMULATION PROGRAMMING & MODSIM

2.1 SIMULATION PROGRAMMING

In the process of developing and implementing a simulation model, an analyst has to decide on the following features needed in programming:

1. Event-oriented Vs Process-oriented Approach
2. Time-Advance Mechanism
3. Random number Generation
4. Collection and Analysis of Data
5. Reporting the Results

Of the above, the first two are discussed here as they are relevant with the present work.

2.1.1 APPROACHES TO DISCRETE-EVENT SIMULATION MODELLING

Almost all simulation languages use one of two basic approaches to discrete-event simulation modelling. These approaches are also used by modellers using a general purpose language.

2.1.1.1 Event-oriented Approach

The classical approach to discrete-event simulation is event-oriented. In this approach, a system is modelled by identifying its characteristic events and then writing a set of event routines which give a detailed description of the state changes taking place at the time of each event. The simulation evolves over time by executing the events in increased order of their time of occurrence [12].

For instance, in a simple queueing model the various routines that describe the events might be:

* Arrival of a customer

* Joining of customer in the queue and waiting
* Customer being served by the server
* Customer leaves the system

The characteristic of this approach is that no time passes during any event routine. Instead, passage of time is handled by scheduling the next event that takes place in the system. For example, in the above model, the event routine "customer being served by the server" would schedule the next event routine, "customer leaves the system" at some future time.

This approach is adequate for smaller models, but in larger models it is often difficult to follow or modify the flow of logic, which describes the behaviour of an object, such as a customer in the queueing model. Consider the simple queueing model with some extra features like reneging, jockeying and some other features performed by customer. There would be many unrelated event routines. Following the logic flow which describes the behaviour of a customer would be like tracing through a sequence of GOTO statements in a large BASIC or FORTRAN program.

2.1.1.2 Process-oriented Approach

A *process* represents an object and the sequence of actions it experiences throughout its life in the model [15]. Hence, a process can be defined as a time-ordered sequence of events, separated by lapses of time, which describes the entire experience of an object in the model, like a "customer as it flows through a queueing system".

In this approach, a system is modeled by writing a process routine which delineates everything that happens to an object as it moves through its corresponding process. Unlike the event-oriented approach, the total history of the object (say, a customer in the queueing model) is described by a single (process) routine which also explicitly contains the

passage of simulation time. The approach is also called 'process-interaction approach', because of the fact that at a given time, a system may contain many processes (of, say, customer) interacting with each other while competing for a set of system resources (server).

A process object enters a model at an explicit simulation time, its "creation time". It becomes active either immediately or at a prescribed "activation time". From there on, the description of its activity is contained in the process routine.

The amount of lapse of time that separates the events of the processes can be either predetermined or indefinite [15]. Predetermined lapses of time are used to model such phenomena as the service time (deterministic or stochastic) whereas indefinite delays arise because of competition between processes for limited resources. The former case is called *unconditional wait* and the latter is known as *conditional wait.* In the latter case, processes will automatically be delayed until the resource is made available to it.

At each (re)activation of the process routine, it may execute statements representing changes to the system state. The process routine may test for system conditions and take alternative courses of action.

Processes interact either implicitly (e.g., through resource competition) or explicitly (through executing statements like "activate", "interrupt", "resume", etc.).

The distinction between event and process approaches can be explained as follows: A process may be thought of as a sequence of events - which, in fact, it is. If we wish to replace even the most elementary process by the equivalent event code, it will require atleast two events whenever we wish to represent a simulated time delay [15]. A small example may serve to clarify this point: Consider a telephone call as process. The

process routine might be coded as below:

```
PROCESS CALL
      Add 1 To Number_of_BusyLines
      Wait For 5 Minutes
      Subtract 1 From Number_of_BusyLines
END
```

An equivalent event model might be as follows:-

```
EVENT BEGIN_CALL
      Add 1 To Number_of_BusyLines
      Schedule The Event END_CALL In 5 Minutes
END

EVENT END_CALL
      Subtract 1 From Number_of_BusyLines
END
```

The process-interaction approach has several advantages over the event-oriented approach. First, for many types of system, it provides a more *natural* framework for modelling because the entire experience of an object as it flows through a system is described in a single process routine. This is particularly of great benefit in the case of large and complex systems with unrelated events of the objects in the system. Furthermore, when this approach is implemented in a simulation language, the language provides powerful 'macrostatements', which automatically translate certain situations commonly occurring in a simulation model, e.g.,customers arriving to a queueing system, into the corresponding event logic. As a result a model written using this approach may require significantly fewer lines than the corresponding model using the event-scheduling approach.

2.1.2 TIME-ADVANCE MECHANISMS

Historically, two principal approaches have been suggested for advancing the simulation clock, namely, *next-event time advance* and *fixed-increment time advance.* The former approach is the one that is used by all major simulation languages, and by most people coding their model in a general-purpose language.

With the next-event time advance approach, the simulation clock is initialized to zero and the times of occurrence of future events are determined. The simulation clock is then advanced to the time of occurrence of the most imminent (first) of these future events, at which point the state of system is updated. Then the simulation clock is advanced to the time of the (new) most imminent event, the state of the system is updated, and future event times are determined, and so on. This process of advancing the simulation clock from one event time to another is continued until some prespecified stopping condition is satisfied eventually. Since all state changes occur only at event times for a discrete-event simulation model, periods of inactivity in a system are skipped over by jumping the clock from event time to event time. It may be noted that the successive jumps of the simulation clock are generally not equal in size.

In the fixed-increment time advance approach, the simulation clock is advanced in the increments of exactly Δt time units for some appropriate choice of Δt. After each update of the clock, a check is made to determine whether any events scheduled have occurred during the previous interval of length Δt. If one or more events were scheduled to have occurred during that interval, these events are considered to have occurred at the end of the interval and the system state (and statistical counters) are updated accordingly. The primary use of this approach appears to be for systems where it can resonably be assumed that all events *actually* occur at one of the times that is a multiple of Δt for an appropriately chosen Δt. For example, data in economic systems are often available on yearly basis, and it is natural in simulation model to advance the simulation clock in increments of one year.

2.1.3 SOME PROCEDURAL SIMULATION LANGUAGES

2.1.3.1 SIMULA

SIMULA is an old simulation language, which was developed by Dahl and Nygaard [8]. It is among a class of simulation languages that translate chosen syntactic structure into an ALGOL base. One of the unique features of SIMULA is the capability to create, destroy and modify existing and new processes created by collections of SIMULA control statements. SIMULA creates a common data file that is accessible to all processes. SIMULA deals with activities that can be created or destroyed through structured groups of SIMULA statements and commands. A transaction can be either created or destroyed by processes; however, the procedure must be constructed by the user for each individual case. Program logic is controlled by a master clock routine, and the language provides pre-programmed logic to connect all the components of the model. It provides a random number generator, several random variable generation schemes, a fixed format output, and error-checking devices.

2.1.3.2 GPSS

GPSS (General Purpose System Simulator) is a simulation language originally developed by G.Gordon [11]. Three major revisions of GPSS are now commonly used. (1) GPSS/V, (2) GPSS/H, and (3) GPSS PC. GPSS/V is a descendent of GPSS/GPSS III and is recognized as the *de facto* standard version of GPSS. The principal appeal of GPSS is the use and speed with which simulation models can be built. Since many simulation projects operate against tight time deadlines, the programming power can be a very important consideration. GPSS in all forms represent a process-orientation conceptualized via block diagrams that possess special attributes, logic and control statements. GPSS constructs a logical model of the system under study through the use of block commands. There are more than 40 different

specialized blocks in GPSS. These commands perform specific functions that are unique to the language. Time is advanced in fixed units as transactions flow through a specified sequence of block commands. Transactions possess certain attributes that can be used to make logical decision at chosen block commands. Each block type might have names, symbols, or numbers associated with it, and each block consumes a specified amount of time to process a transaction. GPSS provides a fixed format output, and collects predestined statistics automatically.

2.1.3.3 SIMSCRIPT II.5

SIMSCRIPT II.5 is an event-oriented or process-oriented simulation language considered by many to be the most powerful simulation language available today. After having been developed by Rand Corporation in the early 1960s, SIMSCRIPT evolved through a number of versions; the latest and most comprehensive version, SIMSCRIPT II.5 is a proprietary product of CACI.

Because of the power and diversity of the statements available in SIMSCRIPT, general programming tasks can be done more efficiently than in FORTRAN. Furthermore, its English-like and free-form syntax makes SIMSCRIPT programs easy to read and almost self-documenting. A SIMSCRIPT model views a system as consisting of *entities, attributes* and *sets.* Entities are of two types, permanent and temporary. Permanent entities correspond to objects in a system, e.g., servers in queueing system, whose number remains fairly constant during the simulation. Conversely, temporary entities represent objects in a system, e.g., customers arriving to a queueing system, whose number may vary considerably during simulation. Attributes are data values which characterize either type of entity, and sets are collections of entities with a common property. To construct a discrete-event simulation in SIMSCRIPT II.5, the modeler must

write a preamble, a main program, and the usual event routines. The preamble, which does not contain any executable statements, is used to give a static description of the entire model, including the required events, entities, and sets. Main program is used to initialize the event list, to initialize state variables, and to read input parameters. SIMSCRIPT II.5 supports process-oriented approach also.

2.2 SIMULATION WITH MODSIM

MODSIM is developed by CACI Products Company, California, whose previous simulation product, SIMSCRIPT II.5 is considered to be the most popular simulation language presently. It has object-orientation capability which makes it distinct from all other simulation languages. It has a user-friendly compiler, MSCOMP, which requires Turbo C compiler to compile MODSIM programs [5]. The various features of MODSIM are explained below.

2.2.1 MODSIM OBJECTS

Objects in MODSIM are dynamically allocated data structures coupled with methods. The fields in the object's data structure define its state at any instant in time while its methods describe the actions which the object can perform.

Object types are declared with the specification of its fields and methods in the TYPE section of the module. This is called *object type declaration.* For each method heading which is mentioned in the object type declaration, there must be a full method declaration in a separate *object declaration.* The object declaration is a separate, named block which contains the implementation of all the methods declared for the corresponding object in its type declaration.

A method differs from a procedure in several important ways:

* A method is tied to an object. It can only be invoked by sending a message to an object instance requesting that the method be performed.

* Unlike procedures, there can be any number of methods named with same identifier in a program. Each one can have different implementation. They can be distinguished from each other because each is tied to a different object type.

* Some methods can elapse simulation time but procedures cannot.

MODSIM methods are of three form: **ASK** method, **TELL** method and **WAITFOR** method. The kind is specified when they are declared. There are important distinctions between ASK, TELL and WAITFOR methods as pertains to simulation.

An ASK call is identical to call and is invoked by ASK statement as shown below:-

ASK object [TO] method_name [(parameter list)]

When the ASK statement is executed, a message is sent to the object requesting it to invoke the method. The calling code then *waits* for the invoked method to finish before proceeding past the ASK statement. ASK methods are not allowed to pass any simulation time, so in a simulation, the action done by an ASK method takes place at one instant of simulation time. In other words, the ASK method call is a *synchronous call.*

The TELL method, which is known as a *delayed method call,* is essentially an asynchronous call which is used only during simulation. The calling code executes the TELL statement (which is same as ASK statement with the word 'ASK' replaced by 'TELL') which sends a message requesting the object to invoke the method. The calling code then proceeds past the TELL statement without waiting for the invoked method to complete execution or, for that matter, even to start. TELL methods are allowed to pass simulation time.

The remaining type of method, the WAITFOR method, is somewhat hybrid between the TELL and ASK methods. Like the TELL method, the WAITFOR method may elapse simulation time. Unlike the TELL method, it may only be invoked by a 'WAIT FOR' statement (to be discussed shortly) and it can return some value to the invoking method. This is possible because the latter will not continue until the WAITFOR method finishes. In this aspect, the WAITFOR method bears some similarities with ASK method.

2.2.2 SIMULATION CAPABILITIES OF MODSIM

MODSIM has powerful and flexible capabilities for dealing with discrete-event simulation. Each object is capable of carrying on multiple, concurrent activities, each of which can elapse simulation time. An activity is an event scheduled by an object instance using a TELL or WAITFOR method which is capable of elapsing simulation time. The activities can operate autonomously or they can synchronize their operation.

Not only an object instance have multiple TELL and/or WAITFOR methods carrying on activities simultaneously with respect to simulation time, but any one method of the object instance can be invoked multiple times.

2.2.2.1 Time-Advance Mechanism of MODSIM

Simulation time is automatically maintained by MODSIM. The units of time used are dimensionless. They can represent whatever granularity of time is appropriate for the simulation and it is upto the user to explicitly perform any unit conversions.

At any instant of simulation time, there can be multiple, concurrent activities. In reality, on traditional sequential computer architectures, the activities which appear to be happening at the same point of simulation time are carried out sequentially by the computer in actual 'wall clock' time. Once all activities scheduled for a particular instant

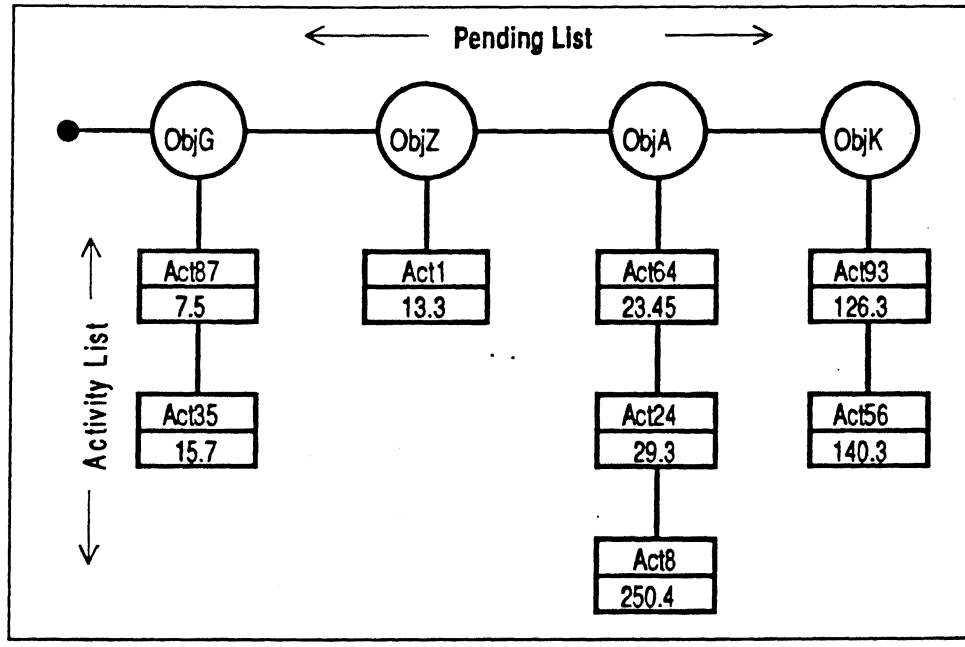


FIGURE 1 : The system's pending list

of simulation time have been carried out, the simulation clock is advanced to the next point in simulation time when an activity has been scheduled.

2.2.2.2 The System's Pending List - Objects' Activity Lists

To keep track of all activities which have been scheduled, MODSIM keeps a *pending list.* The pending list is an ordered list of all objects which have scheduled activities. The object with the most imminent activity is ordered first in the list. Each object instance, in turn, keeps its own list of activities which it has scheduled. The object instance's activity list is ordered by the most imminent activity. Figure 1 shows this two-dimensional structure.

2.2.2.3 Process-oriented Approach of MODSIM

MODSIM strongly supports process-oriented discrete event simulation. The process approach simplifies larger models by allowing all

of the behaviour of an object in a model to be described in one or more TELL and/or WAITFOR methods which allow for the passage of time at one or more points in the method. There is a further advantage to the process technique with MODSIM. Once the actions of a class of objects (such as customers in the queueing model) have been gathered together in an object, the simulation program can create multiple, concurrent instances of that object. In the queueing model, for example, the simulation program would generate a new instance of the customer object each time a customer arrived. While there would be multiple, distinct copies of the customer object operating simultaneously, each could have different values of their fields to describe the particular customer's properties. Finally, process objects can interact in the natural way. In a waiting line, an instance of the customer object with the young attribute might yield its place in the queue to a customer object with elderly attribute!

A simulation model written in MODSIM defines a system in terms of processes because the process technique provides a powerful structure for expressing most categories of the simulation problems, and provides significant advantages over the direct use of discrete events.

The advantages of processes are both conceptual and labour-saving. The process statements are expressed sequentially, in a manner analogous to the system being described. This practice is recommended by standard design methodologies.

2.2.2.4 Time Elapsing Methods - the WAIT Statement

The time elapsing TELL or WAITFOR method is the construct which supports this process-oriented approach to simulation. In these methods, it is possible to use a WAIT statement to indicate that the simulation time should elapse at some point or points in the method. Each WAIT statement in the method is considered to be an activity of the parent

object. When a WAIT statement is encountered, the TELL or WAITFOR method suspends execution. When the specified amount of simulation time has elapsed, the method resumes execution. We say that object has carried out an activity.

A WAIT statement specifies the reason for the wait, a sequence of events to be executed after the WAIT is successfully completed and an optional sequence of statements to be executed if the WAIT is *interrupted.* The syntax of WAIT statement is:

```
WAIT reason
      Statement Sequence
[  ON INTERRUPT
      Statement Sequence ]
END WAIT;
```

where *reason* is a keyword, *DURATION* or *FOR*, followed by any required identifiers. The ON INTERRUPT clause is optional. If the wait is 'successful', the first statement sequence is executed, otherwise the sequence after the ON INTERRUPT is executed instead. In either case, execution continues after the END WAIT unless one of the statement sequences alters the flow of control.

A WAIT statement can appear in a TELL or WAITFOR method. The most basic WAIT is one for a specific period of time. A wait for a specified period of simulation time is achieved with WAIT DURATION clause. There are two other variations of the WAIT statement discussed shortly. One allows the involved method to wait until another method which is invoked completes execution. Another variation allows the method to wait until some triggering event occurs.

2.2.2.5 Synchronizing Activities

In some simulation scenarios, two activities must operate synchronously. One activity starts a second activity then suspends its execution and waits over a period of simulation time for the second activity to complete before it resumes execution. To accomplish this, MODSIM

provides the *WAIT FOR* statement, the syntax of which is:

```
WAIT FOR object [TO] method[ (argu) ]
      Statement Sequence
[ ON INTERRUPT
      Statement Sequence ]
END WAIT;
```

WAITFOR methods are a special case of the TELL method developed specifically for this case. They allow user to exploit the fact that the invoking method will not proceed until the method being waited for returns.

2.2.2.6 Arbitrary Synchronization with Triggering Objects

Some processes will need to wait until a specified condition occurs. For these cases, MODSIM library provides a special object type, TriggerObj, which along with the WAIT FOR statement, allows a method to pause and wait until some condition occurs. The syntax of the statement is:

```
WAIT FOR TriggerObj [TO] Fire
      Statement Sequence
[ ON INTERRUPT
      Statement Sequence ]
END WAIT;
```

When the WAIT FOR ... Fire statement is encountered, the methods suspends and waits until the trigger object is triggered by some other method. At that time the first statement sequence set is executed. If the trigger object interrupts the method, the other sequence if executed. A trigger object can have any number of methods waiting for it to trigger.

2.2.2.7 Multiple Process Activities

To construct realistic simulation models, it is often necessary to model a physical object which can perform several operations simultaneously. A tank in a ground combat model, for instance, may be required to perform movement, communications and target acquisition activities simultaneously. Although this is a fairly common situation, it has traditionally been difficult to model, particularly when the activities may interact. To support such models, MODSIM allows an object to do more than one thing at once. For example, a process object may be in the middle

of one operation when it receives a message to perform a different, conflicting operation. In response, the object can:

* Interrupt the conflicting time-elapsing method which is waiting
* Ignore the new request
* Defer the new request.

2.2.2.8 Interrupting Activities

MODSIM has provisions for interrupting and stopping any or all activities prematurely. Any time-elapsing method can be interrupted by invoking the built-in *interrupt* procedure which takes as parameters the object reference value of the object to be interrupted and the name of the particular method to be interrupted.

2.2.2.9 Grouping Objects

A language which makes use of dynamic data structures, such as objects needs a way to group related objects in a disciplined way. This is especially true for simulation, which typically group objects queueing for a resource. Such associations are referred to as groups in MODSIM. Objects may be selectively added to or removed from a group. A MODSIM program can iterate through a group examining the members of that group. Groups are *untyped* so that they can hold a mixture of object types. An object can belong to any number of groups.

When groups are used, the ordering may be implicit or explicit. There are a variety of groups available: queue, stack, ranked, btree. Since these groups are provided as objects, the user can easily derive their own object type and modify the behaviour as required.

2.2.3 OTHER KEY FEATURES OF MODSIM

2.2.3.1 Modular Structure

One of the MODSIM's strong points is its modular structure which allows programs to be constructed from library modules. Any part of a

program can import types, variables, constants, data structures like objects, and procedure definitions from these library modules as needed.

There are three types of MODSIM modules: *main, definition* and *implementation* modules. Every MODSIM program must contain a main module. As the name implies, there can only be one main module in a program. Each module is named using a standard identifier. A program may contain any number of modules. Each module is stored in a separate file. Any module can be compiled separately.

A library consists of two modules: definition and implementation. *Each is named with the same identifier.* Any constant, type, variable, or procedure declared in a definition module can be imported by other modules. All the contents of a definition module is implicitly visible in the accompanying implementation module. However, nothing in an implementation module is visible anywhere else, including within that library's definition module. Hence, nothing can be imported from an implementation module. Also, nothing can be imported from a main module. Anything imported in a definition module is not implicitly visible in that library's implementation module. There can be no executable code in the definition module. Also, if a procedure or object method is defined in a definition module, it must be coded in the accompanying implementation module.

Modular structure has significant gains in terms of reduced development time, program maintenance and reusability of code.

2.2.3.2 Inheritance

In MODSIM, a new object can be defined in terms of existing object type. This is called inheritance. The capability to inherit the fields and methods of an object and elaborate on them is a powerful feature. If the language were to impose traditional type rules on the objects

involved in an inheritance hierarchy, this would limit the usefulness of objects.

Figure 2 demonstrates an inheritance hierarchy of objects all descended from one common object, the *PoweredObj*. If the Figure is viewed from the perspective of the *AircraftObj*, the relationship of the objects in the tree can be described as follows: All of the types types above *AircraftObj* in the tree are called *underlying types.* The immediate predecessor, VehicleObj, is called *base type.* All descendents of *AircraftObj* are called *derived types* of *AircraftObj*

The hierarchical type rules for the objects state that a reference value for an object can safely be assigned to a reference variable of one of its underlying objects. The converse, however, is not true.

The derived type will typically define additional fields and/or methods not present in the base type. It may also *override* the implementation of a method defined in an underlying object type and replace it with its own. Any method not explicitly overridden by the derived type is automatically inherited from the base type. Similarly the fields are

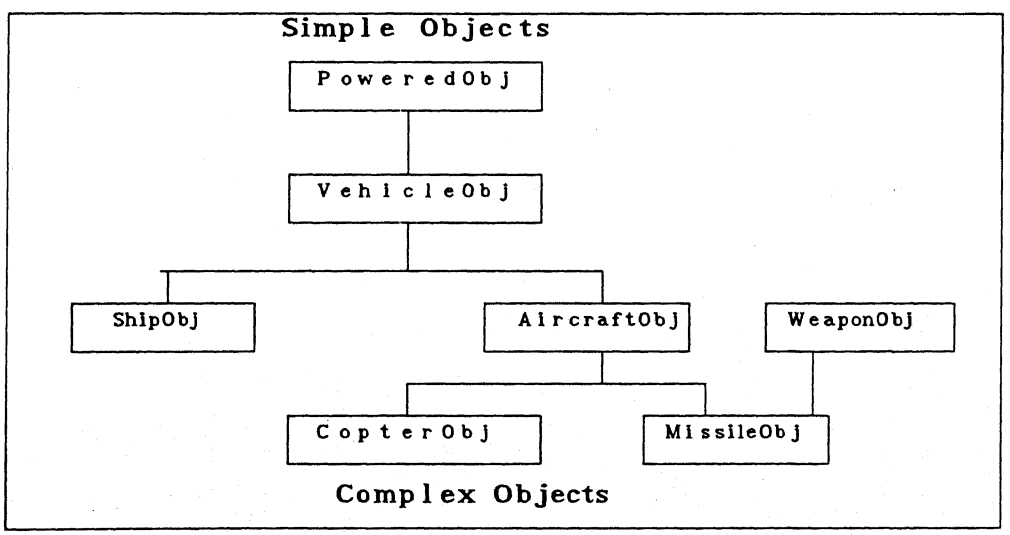


FIGURE 2 : Object Type Hierarchy

also inherited. A overriding method in a derived type can invoke a method of the same name in an underlying or base type. This is useful when the programmer wishes to append new code to an old method of the same name.

MODSIM also supports a form of inheritance known as *multiple inheritance* in which a new object type is defined in terms of two or more existing object types. When a new object type is defined in this way, it has a copy of each field and each method of each of its base types. *MissileObj* of Figure 2 demonstrates multiple inheritance. The existing *AircraftObj* along with the *WeaponObj* creates a complex object *MissileObj*, which will have all the fields and methods of both *AircraftObj* and *WeaponObj* besides defining its own new fields and methods.

CHAPTER 3

SIMULATION MODELLING & IMPLEMENTATION

Though building simulation model has a systematic procedure defined, the programming style largely depends upon the programmer. Complex models can be implemented efficiently and simple can be made complex. However, availability of powerful data structures in the language ultimately decides the efficiency of the code. Based on this premise, the present work makes an attempt to compare traditional approach with the approach adopted by MODSIM, the simulation language with object-orientation.

In order to describe the process of building the selected simulation models using MODSIM through object-oriented analysis, a step-by-step development of the model is carried out, adding various features at each step. The models selected are queueing and inventory systems. As the focus of the work is on the simulation programming of a system than the system itself, the models are typical in nature for simulation, yet complex enough to represent the real systems. Also, traditional modelling is described upto a certain level.

3.1 QUEUEING MODELS

3.1.1. MODEL DESCRIPTION

The queueing model developed is multi-server, multi-queue and multi-kind customer model with the customer features of *reneging* and *jockeying*. The servers can be of permanent type (i.e., servers last till the simulation is over) or of dynamic type (i.e, they can be created when there is a demand and disposed when idle) or of both type. The model is of multi-kind customer in the sense that customers are of two or more types and each server is earmarked for a kind of customer. The feature of reneging is displayed by the customer by leaving the system if the waiting time exceeds

a particular limit. Jockeying is that property by which if a customer in a multi-server queue finds another queue with smaller length, he jumps to the shorter queue.

The model is developed in stages with addition of features in steps to facilitate a systematic analysis of the model programming.

3.1.2 MODEL DEVELOPMENT & IMPLEMENTATION

3.1.2.1 Simplest Model

At first, consider the simplest queuing model, a single-server model. In traditional modelling, with event-oriented approach, the events for the system can be identified as:

* Arrival of a customer to the system
* Departure of a customer from the system after completing service

As an example, the flowchart of second event is shown in Figure 3. The implementation in traditional approach typically consists of the event routines (based on the flowcharts) besides the sub-routines for initializing the simulation model, for generating the random-numbers (in case of a general purpose language), and for reporting the results. The whole model is run by a main program which typically controls the invocation of the sub-routines.

In this way, the computer model has no direct representatives of the real-world entities like the server and the customer. The modeller has to decide on the various events in the system, various parameters of the model and other programming considerations. Therefore, the modelling approach is not a direct replication of the real-world system, but is a sort of mapping of the real-system into various data structures of the computer model.

In contrast, the approach required in object-oriented analysis is more *natural*. The major task of the modeller here would be to

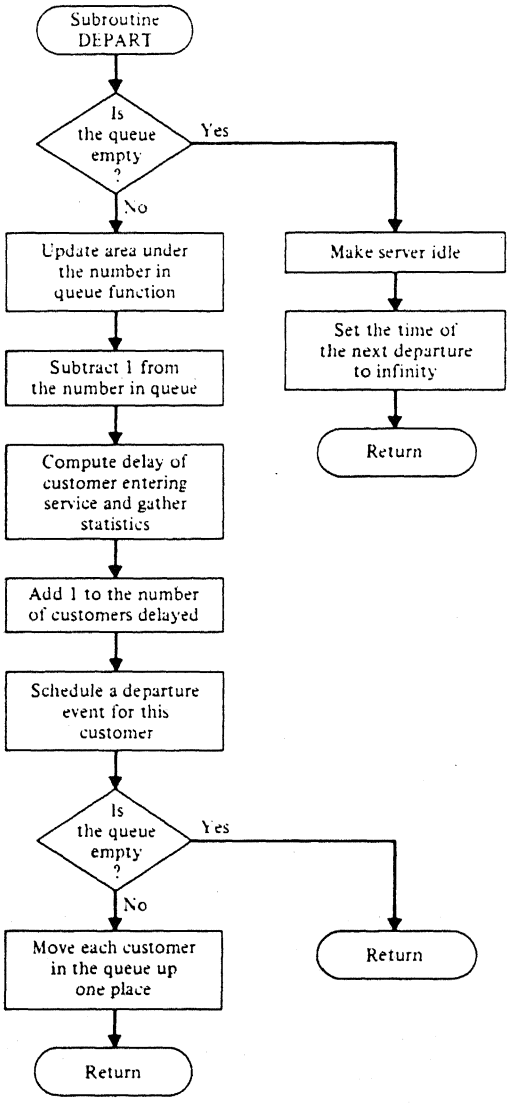


FIGURE 3: FLOW CHART FOR DEPARTURE EVENT ROUTINE

identify the "objects" of his model. Hence, in the present model the objects would be:

* The Server Object and

* The Customer Object

However, the tricky part of the approach is to identify the "logical" objects of the system, i.e., objects which are not directly apparent in the system, but are essential constituents of the system, like, in the present case, the waiting queue of the system. Hence, we have one more object:

* Waiting Queue Object

Besides the above objects, we need to have supporting objects that help drive the simulation model. In this case, an object is needed that generates the customer objects at various simulation times and disposes them as they complete their role in the model and also checks the terminating conditions for the simulation. So, we have a:

* Generator Object

Once objects are identified, the next task would be to define each object's *properties,* i.e., its fields and methods. Fields are the data about the object and methods are the activities that can be carried out by the object. They are discussed later. Assuming that objects and their properties are identified, it can now be explained as to how exactly the simulation is carried out by MODSIM.

It may be recalled that this style of modelling is made feasible by the process-oriented approach. Hence, by defining the objects, we implicitly define the processes of the model, in this case four, *viz.,* the processes for customer, the server, the queue, and the generator object. To illustrate the point more succinctly, consider Figure 4, which describes the customer-process routine. This process routine describes the entire

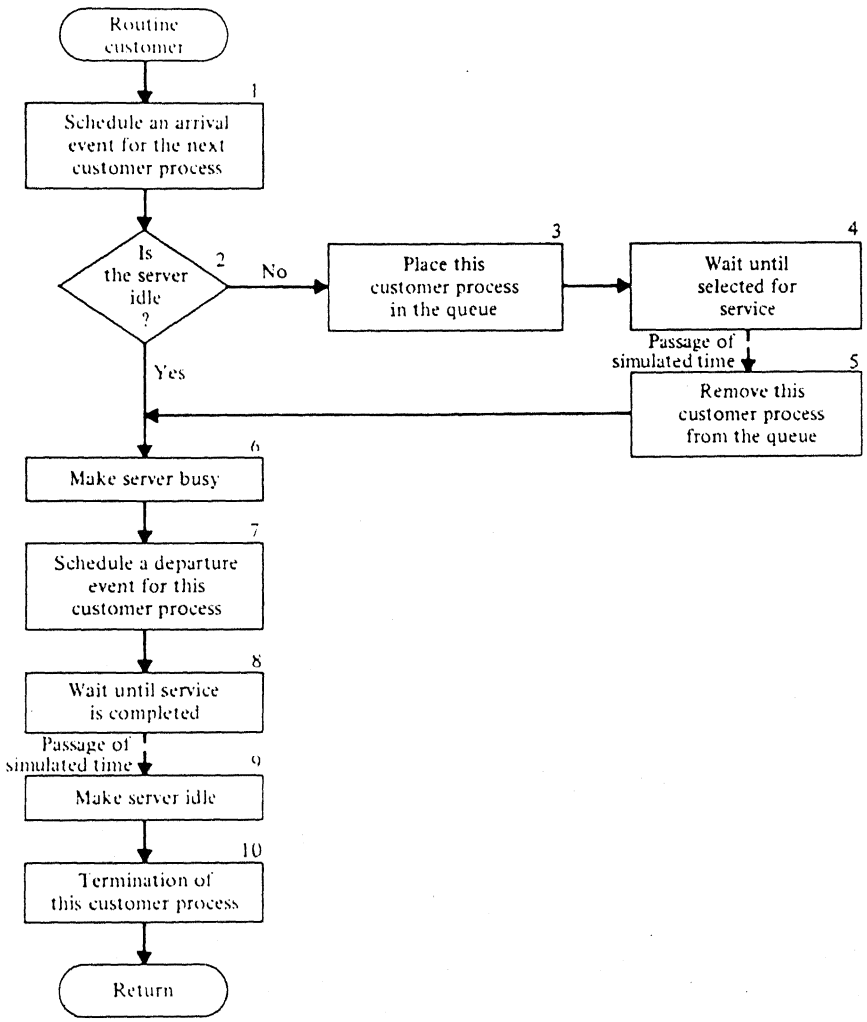


FIGURE 4: CUSTOMER-PROCESS ROUTINE

experience of a customer as it progresses through the system. Unlike an event routine, this process routine has multiple entry points (at blocks 1, 5 and 9). Entry into this routine corresponds to arrival of a customer. To determine whether the currently arriving customer process can begin service, a check is made (at block 2) to see whether the server is idle. If the server is busy this customer process is placed at the end of the queue (block 3) and made to wait (at block 4) until selected for service at some undetermined time in the future. (This is a conditional wait.) Control is then passed to an appropriate customer object to execute one of its waiting activities by MODSIM (with the help of pending list). When this customer process (the one made to wait at block 4) is activated at some point in the future (when another customer process completes service and makes the server idle), it is removed from the queue (at block 5) and begins service immediately, thereby making the server busy (block 6). At block 7, the departure time for the customer process beginning service is determined and added to the object's activity list. This customer process is then made to wait (at block 8) until its service has been completed (An unconditional wait). Control is then returned to MODSIM to determine what customer process will be processed next. When the customer process made to wait at block 8 is activated at the end of its service, it makes the server idle at block 9, allowing first customer in the queue to be activated for service. This completes the customer process.

The above description of customer object's process-routine can be extended to other objects. In this way, all objects have activities that elapse simulation time. MODSIM completely takes care of the scheduling and activating (or interrupting) of various activities of the all the objects in the model and hence runs the simulation clock.

The determination of the *properties* of the objects is felt to

be the most critical aspect of simulation modelling through object-orientation. It has significant impact on the validity of the model, as mistakes in even one of the properties of an object may make entire model invalid. Also, it has a considerable bearing on the ease with which the model can be developed. Besides, it has great effect on the *reusability* and *maintainability* of the code in the model.

Deciding on the fields of the object is generally straightforward. However, deciding on the methods needs thorough understanding of the working of the system. The key point lies in deciding the *type* of the method: ASK method or TELL method or WAITFOR. One direct distinction is that ASK method cannot elapse simulation time, while the other two can (explained in previous chapter). Also, when the method is required to return some value to the invoking code, it can be ASK method only. However, TELL methods need not *necessarily* elapse simulation time. Hence, choosing between ASK and TELL methods should be done judiciously.

Considering the Server object defined earlier, the fields required are: the value indicating the total busy time of the server, maximum service time, the status (idle or busy) and a random number generator (in the form of *RandomObj*, defined in the standard library). The methods could be: one to supply service time to customer, and one to complete the service and engage the next customer. In this case, both methods happen to be ASK methods.

Customer object needs a field that indicates its arrival time and a method to engage the server as the service starts. As this method has to elapse the simulation time in the form of service time, it can be TELL method only.

The Queue object is essentially the queue object type defined in the standard library with minor modifications to accommodate for some

statistics like maximum waiting time and maximum number waiting in the queue. Hence, this object *inherits* the library object and *overrides* some of its methods to take care of the additional statistics.

Finally, the Generator object has a counter for number of customers produced during simulation. And a TELL method that drives the simulation over time.

The model works like this: The Generator object is sent a *message* from the main routine to start the simulation. The former starts generating the customers, which in turn either start being served or join the queue. Whenever a customer completes the service, it invokes the servers method to finish service, by which the next customer takes over the server. The customers served are eventually disposed by the Generator object. Once the simulation is terminated, the main module invokes the routine that reports the results. The entire code in MODSIM is as follows:

```
MAIN MODULE SingleServer;

{First, import various objects and procedures from standard library.}
FROM SimMod IMPORT StartSimulation, SimTime;
FROM RandMod IMPORT RandomObj, FetchSeed;
FROM GrpMod IMPORT QueueObj;

TYPE
      ServerStatusType = (Idle, Engaged);

WaitQObj =                          {Declaration of Queue object}
OBJECT(QueueObj)                    {Inherits the 'QueueObj' of standard Library}
  maxWaitNum   : INTEGER;
  maxWaitTime : REAL;
  OVERRIDE                          {Override the following inherited methods}
    ASK METHOD Add(IN cust : ANYOBJ);
    ASK METHOD Remove() : ANYOBJ;
END OBJECT;

ServerObj =                         {Declaration of Server object}
OBJECT(WaitQObj)
  totalBusyTime,                    {Proportion of Time Engaged}
  maxServiceTime   : REAL;
  status     : ServerStatus;
  ranGen     : RandomObj;           {Random number Generator}
  ASK METHOD ObjInit;               {Object initialization method}
  ASK METHOD ServeCust() : REAL;
  ASK METHOD FinishService;
```

```
END OBJECT;

CustomerObj =                    {Declaration of Customer object}
OBJECT
  startWaitTime  :REAL;          {Arrival time}
  TELL METHOD EngageServer;
END OBJECT;

GeneratorObj =                   {Declaration of Generator object}
OBJECT
  numcustomers : INTEGER;
  TELL METHOD GenCustomers(IN meanInterArrival,simDuration : REAL);
END OBJECT;

VAR                {These are global variables.}
  hoursToRun, meanInterArriveTime, meanServiceTime  : REAL;
  waitQ : WaitQObj;
  server : ServerObj;
  CustGenerator : GeneratorObj;

{Implementation of the methods of the objects start here.}
OBJECT CustomerObj;
  TELL METHOD EngageServer;
  VAR
    serviceTime : REAL;
  BEGIN
      IF ASK Server status = Engaged          {Server engaged, so Q up and wait}
        ASK waitQ TO Add(SELF);
        startWaitTime := SimTime();
        TERMINATE;
      END IF;
      serviceTime := ASK server TO ServeCust();
      WAIT DURATION serviceTime;          {wait for service duration}
      END WAIT;
      ASK server TO FinishService;
      DISPOSE(SELF);          {leave the model}
  END METHOD;
END OBJECT;

OBJECT ServerObj;
  ASK METHOD ObjInit;
  BEGIN
      NEW(ranGen);
  END METHOD;

  ASK METHOD ServeCust() : REAL;
  VAR
    svcTime : REAL;
  BEGIN
      status := Engaged;
      svcTime := ASK ranGen Gamma(meanServiceTime, 2.0);
      totalBusyTime := totalBusyTime + svcTime;
      IF svcTime > maxServiceTime
            maxServiceTime := serviceTime;
      END IF;
      RETURN serviceTime;
  END METHOD;
```

```
  ASK METHOD FinishService;
  VAR
     nextCust : CustomerObj;
  BEGIN
     status := Idle;
     IF ASK waitQ numberIn > 0                {check if anybody in the queue}
           nextCust := ASK waitQ TO Remove();
           TELL nextCust TO EngageServer;
     END IF;
  END METHOD;
END OBJECT;

OBJECT GeneratorObj;
  TELL METHOD GenCustomers(IN : interArriveTime, numHours : REAL);
  VAR
     ranNumGen : RandomObj;
     waitTime : REAL;
     customer : CustomerObj;
  BEGIN
     NEW(ranNumGen);
     ASK ranNumGen TO SetSeed(FetchSeed(3));
     WHILE SimTime() < hoursToRun * 60.0
           waitTime := ASK ranNumGen Exponential(meanInterArriveTime);
           WAIT DURATION waitTime;      {wait for arrival of the next customer}
           END WAIT;
           NEW(customer);
           INC(numCustomers);
           TELL customer TO EngageServer;
     END WHILE;
     DISPOSE(ranNumGen);
  END METHOD;
END OBJECT;

OBJECT WaitQObj;
  ASK METHOD Add(IN obj:ANYOBJ);
  BEGIN
     INHERITED Add(obj);
     IF numberIn > maxWaitNum
           maxWaitNum := numberIn;
     END IF;
  END METHOD;

  ASK METHOD Remove() : ANYOBJ;
  VAR
     cust : CustomerObj;
  BEGIN
     cust := INHERITED Remove();
     IF maxWaitTime < (SimTime()- ASK cust startWaitTime)
           maxWaitTime := (SimTime()- ASK cust startWaitTime);
     END IF;
     RETURN cust;
  END METHOD;
END OBJECT;

PROCEDURE ReportResults;
BEGIN
```

```
        OUTPUT("Parameters of the system are:");
        OUTPUT("Inter arrival time", meanInterArriveTime);
        OUTPUT("Mean Service Time", meanServiceTime);
        OUTPUT("Simulation Duration:", hoursToRun);
        OUTPUT("Simulation Results");
        OUTPUT("Stop Time in hours:", SimTime()/60.0);
        OUTPUT("Number of customers", ASK custGenerator numCustomers);
        OUTPUT("Maximum Q length :", ASK waitQ maxWaitNum);
        OUTPUT("Longest wait time:", ASK waitQ maxWaitTime);
        OUTPUT("Server busy %", ASK server totalBusyTime/SimTime() * 100.0);
        OUTPUT("Longest Service Time:", ASK Server maxServiceTime;
        OUTPUT("    Mean    service    Time:",    ASK    server    totalBusyTime/    ASK
                    cusGenerator numCustomers));
END PROCEDURE;
```

{Main module starts here.}

```
BEGIN
        OUTPUT("Please Enter: Mean inter arrival time in mts.");
        OUTPUT("                    Mean service time in mts.");
        OUTPUT("                    Simulation Duration in hours");
        INPUT(meanInterArriveTime, meanServiceTime, hoursToRun);
        NEW(waitQ);
        NEW(server);
        NEW(custGenerator);
        TELL custGenerator TO GenCustomers(meanInterArriveTime, hoursToRun);
        StartSimulation;
        ReportResults;
END MODULE.
```

The simulation time in the model passes at two points: one by the 'Generator' object, to wait between interarrivals of the customers, and the other by the customer object, during service. In both the contexts, the simulation time is elapsed by using the *WAIT DURATION* statement as below:

```
WAIT DURATION interArriveTime/servicetime
END WAIT;
```

3.1.2.2 Developed Model

Once the basic model is built, extending the model is not a difficult task, as the object-oriented capabilities include *incremental problem solving*, which is appropriate here. The following stages describe how the simple single-server model can be developed into the model specified at the beginning of this chapter.

A. Multi-server single-queue model:

This is essentially same as the previous model, except that

number of servers can be one or more. Hence, we have a queue object data type as a global variable, which at any instant holds idle servers at that instant.

```
VAR
  idleServers : QueueObj;
```

Whenever a customer is created, a check is made for availability of idle servers, and is processed accordingly. Hence, the 'EngageServer' method changes as shown below:

```
TELL METHOD EngageServer;
VAR
  serviceTime : REAL;
  myServer : ServerObj;
BEGIN
   IF ASK idleServer numberIn = 0          {No server free, so Q up}
     ASK waitQ TO Add(SELF);
     startWaitTime := SimTime();
     TERMINATE;
   END IF;
   myServer := ASK idleServers TO Remove();          {Pick an idle server}
   serviceTime := ASK myServer TO ServeCust();
   WAIT DURATION serviceTime;
   END WAIT;
   ASK myServer TO FinishService;
   DISPOSE(SELF);
END METHOD;
```

The 'FinishService' method of the Server Object looks as below:

```
ASK METHOD FinishService;
VAR
   nextCust : CustomerObj;
BEGIN
   IF ASK waitQ numberIn > 0
        nextCust := ASK waitQ TO Remove();
        TELL nextCust TO EngageServer;
   END IF;
   ASK idleServers TO Add(SELF);          {Join the idle server queue}
END METHOD;
```

And the main module is changed as below to create the number of servers specified:

```
FOR k:= 1 TO numServers
     NEW(server);
     ASK idleServers TO Add(server);
END FOR;
```

The procedure that reports the results is also modified accordingly.

B. Multi-server multi-queue model:

This model demands slight restructuring of the objects, though the basic objects continue to be the same. Firstly, as there is a one-to-one mapping between the server objects and the queue objects, we can think of integrating the two, creating more complex objects to generate code with much ease. Here, the *inheritance* power comes into picture. The server object now inherits the queue object of the previous case and is declared accordingly.

```
ServerObj =
OBJECT(WaitQObj)
      .....          {Fields & methods as before}
END OBJECT;
```

And in the rest of the code wherever variable of WaitQ type comes, it should be replaced by the variable of Server type. Further, we need a queue to store the engaged servers:

```
VAR
  busyServers : QueueObj;
```

The customer object now requires one more method: it has to pick a queue or server (the shortest queue) as it enters the system. Hence it requires a field of server type, which is assigned by a method 'PickQueue()' which returns the shortest queue at that instant. The *data-hiding* feature of the MODSIM can be demonstrated here by declaring the method 'PickQueue' as a *private* type, which means that this method can not be invoked by the other objects, and can only be invoked by the object's other methods.

```
ASK METHOD PickQueue() : ServerObj;
VAR
      thisnum, shortestNum : INTEGER;
      thisServer, shortestServer : ServerObj;
BEGIN
      shortestNum := MAX(INTEGER);
      thisServer := ASK BusyServers First();
      WHILE thisServer <> NILOBJ
            thisNum := ASK thisServer numberIn;
            IF shortestNum > thisNum
```

```
                shortestNum := thisNum;
                shortestServer := thisServer
            END IF;
            thisServer := ASK busyServers Next(thisServer);
        END WHILE;
        RETURN shortestServer;
END METHOD;
```

The integration of the waiting queue object necessitates the customer object to execute a *conditional wait* until its turn comes for the service. This essentially is arbitrary synchronization with a trigger object, which triggers when the customer's turn comes for the service.

```
  TELL METHOD EngageServer;
  VAR
    serviceTime : REAL;
  BEGIN
      IF ASK idleServer numberIn = 0
            myServer := PickQueue();
            ASK myServer TO Add(SELF);
            startWaitTime := SimTime();
            WAIT FOR TriggerObj TO Fire;
            END WAIT;
      ELSE
            myServer := ASK idleServers First();
      END IF;
      serviceTime := ASK myServer TO ServeCust();
      WAIT DURATION serviceTime;
      END WAIT;
      ASK myServer TO FinishService;
      DISPOSE(SELF);
  END METHOD;
```

To trigger the waiting object, the Customer object needs a method:

```
ASK METHOD LeaveQ;
VAR
      me : CustomerObj;
BEGIN
      me := ASK myServer TO Remove();
      TELL myTrigger TO Trigger;
END METHOD;
```

In the remaining aspects, the model retains the code of the previous stage.

C. Multi-server multi-queue and multi-kind customer model:

In this model, the customers (and hence servers) are classified into many types. However, for the model developed, only two

types are assumed (divided as gent and lady customers). The data abstraction capability comes in handy in this case in the form of the customers type (gent or lady) becoming a field of the customer's object and is assigned by one of its own methods, 'AssignSexAndQType' given below.

We now need a queue each as global variables to store idle and busy servers, for each type of customer/server.

```
VAR
IdleLadyServers,busyLadyServers,IdleGentServers,
busyGentServers: QueueObj;
```

Further, we need two sets of queue object types as fields of the customer object to identify its set of idle servers and busy servers. Similarly, every server also needs two fields to identify it's holder when it is busy and when it is idle. These assignments are made as soon as the respective objects are created.

```
SexType = (Gent, Lady);
ASK METHOD AssignSexAndQType(IN num:REAL) : SexType;
BEGIN
     IF num < 0.5      {This number is generated elsewhere by a Random object}
          sex := Lady;
          myIdleServers := IdleLadyServers;
          myBusyServers := busyLadyServers;
     ELSE
          sex := Gent;
          myIdleServers := IdleGentServers;
          myBusyServers := busyGentServers;
     RETURN sex;
END METHOD;
```

Except for these additions, this model also makes full use of the code developed in the previous stages.

D. Multi-server multi-queue and multi-kind customer model with reneging:

The power of object orientation is more obvious when the ease with which the real-world features can be incorporated in the model is felt. Consider the case of reneging characteristic of the customer. This can be simply achieved by adding one more method to the existing customer object and with minor modifications to the other methods of the customer object.

The new method *renege* schedules an activity at the end of reneging time such that when the customer waits for the stipulated renege time, its trigger object (mentioned previously) is interrupted to make the waiting of the customer for service unsuccessful. This is followed by interrupting all the scheduled activities of the customer and subsequently disposing the customer object in question. However, when the customer is called for service before renege duration the scheduled renege activity is interrupted and customer gets ahead with the service. Hence, the WAIT statement in the method *renege* appears like this:

```
WAIT DURATION renegeTime
ON INTERRUPT          {If the service turn comes before renege time..}
  TERMINATE           {Abort the method}
END WAIT;
```

Also, the conditional wait for the service is modified as below:

```
WAIT DURATION Trigger object TO Fire
ON INTERRUPT          {If waiting is interrupted by renege method..}
  TERMINATE           {Abort wait}
END WAIT;
```

```
TELL METHOD Renege;
VAR
      renegeTime;
BEGIN
      IF sex := Gent
            renegeTime := 30.0;
      ELSE
            renegeTime := 60.0;
      END IF;
      WAIT DURATION renegeTime
      ON INTERRUPT          {If the service turn comes before renege time..}
        TERMINATE           {Abort the method}
      END WAIT;
      ASK myServer TO removeThis(SELF);
      ASK myTrigger TO InterruptTrigger;
      IF numActivities(SELF) = 0
            DISPOSE(SELF);
      END IF;
END METHOD;
```

The 'EngageServer' method now becomes:

```
  TELL METHOD EngageServer;
  VAR
    serviceTime : REAL;
```

```
BEGIN
    IF ASK idleServer numberIn = 0
          myServer := PickQueue();
          ASK myServer TO Add(SELF);
          startWaitTime := SimTime();
          Renege;
          WAIT FOR TriggerObj TO Fire
          ON INTERRUPT
                TERMINATE;
          END WAIT;
    ELSE
          myServer := ASK idleServers First();
    END IF;
    Interrupt(SELF, "Renege");
    serviceTime := ASK myServer TO ServeCust();
    WAIT DURATION serviceTime;
    END WAIT;
    ASK myServer TO FinishService;
    DISPOSE(SELF);
END METHOD;
```

E. Multi-server multi-queue and multi-kind customer model with reneging and Jockeying:

Jockeying is a feature of practical importance. Jockeying is that property by which if a customer in a multi-server queue finds another queue with smaller length at any instant of time, he jumps to the shorter queue. This can be incorporated into the model by introducing a new method each for the customer object and the server object. The Server object has the following method to receive customers by jockeying. This method is executed by the server whenever it completes service to a customer (who leaves the system after that).

```
ASK METHOD Jockey;
VAR
      thisNum, shortestNum : INTEGER;
      thisServer : ServerObj;
      cust : CustomerObj;
BEGIN
      shortestNum := numberIn;
      thisServer := ASK myBusyServers First();
      WHILE thisServer <> NILOBJ
            thisNum := ASK thisServer numberIn;
            IF shortestNum < (thisNum - 1)
                  cust := ASK thisServer Last();
                  ASK cust TO DoJockeying(SELF);
                  EXIT;
            END IF;
```

```
            thisServer := ASK myBusyServers Next(thisServer);
      END WHILE;
END METHOD;
```

For Customer Object, we have:

```
ASK METHOD DoJockeying(IN ser : ANYOBJ);
VAR
      oldServer : ServerObj;
BEGIN
      ASK myServer TO RemoveThis(SELF);
      oldServer := myServer;
      myServer := ser;
      ASK myServer TO Add(SELF);
      ASK oldServer TO Jockey;
END METHOD;
```

F. Multi-server multi-queue and multi-kind customer model with reneging and Jockeying and with dynamic servers:

The system may be designed such that if the queue length exceeds a threshold limit, new servers are created and called into service, and are disposed when they become idle. This is a powerful feature and is difficult to implement in traditional languages. It is implemented in the following manner: A new method is added to the Generator object, which is invoked by the customer object when the number in its queue crosses the limit. This creates a new object of server type (subject to availability of the same). Once the server is created, customers from other queues start jockeying. The server exists as long as there is a queue and is disposed when the server becomes idle.

```
TELL METHOD GenExtraServer(IN cus : CustomerObj);
VAR
      extraServer : ServerObj;
BEGIN
      IF ASK cus sex = Gent
            IF numGentExtra < extraGentServers
                  INC(numGentExtra);
                  NEW(extraServer);
            ELSE
                  OUTPUT("NO MORE EXTRA SERVERS AVAILABLE");
                  RETURN;
            END IF;
      ELSE
            IF numLadyExtra < extraLadyServers
                  INC(numLadyExtra);
```

```
            ELSE
                 OUTPUT("NO MORE EXTRA SERVERS AVAILABLE");
                 RETURN;
            END IF;
      END IF;
      ASK (ASK extraServer myBusyServers) TO Add(extraServer);
      ASK cus TO DoJockeying(extraServer);
      ASK cus TO LeaveQ;
END METHOD;
```

At the end, if we browse through the structure of various objects in the model, the fields and methods contained by various objects are as follows:

```
WaitQObj =                    {Declaration of Queue object}
OBJECT(QueueObj)              {Inherits the 'QueueObj' of standard Library}
  maxWaitNum  : INTEGER;
  maxWaitTime : REAL;
  OVERRIDE                    {Override the following inherited methods}
    ASK METHOD Add(IN cust : ANYOBJ);
    ASK METHOD Remove() : ANYOBJ;
END OBJECT;

ServerObj =                   {Declaration of Server object}
OBJECT(WaitQObj)
  totalBusyTime,              {Proportion of Time Engaged}
  maxServiceTime  : REAL;
  status    : ServerStatus;       {Server type: whether static or dynamic}
  ranGen    : RandomObj;        {Random number Generator}
  numServed : INTEGER;
  myIdleServers,
  myBusyservers  : QueueObj;      {Queue of idle and busy servers at a point}
  ASK METHOD ObjInit;         {Object initialization method}
  ASK METHOD SetStatus(IN mysex :SexType);          {Set if static or dynamic }
  ASK METHOD ServeCust() : REAL;
  ASK METHOD FinishService;
  ASK METHOD Jockey;

END   OBJECT;

CustomerObj =                 {Declaration of Customer object}
OBJECT
  sex  : SexType;
  startWaitTime  :REAL;       {Arrival time}
  myTrigger : TriggerObj;
  myServer  : ServerObj;
  myIdleServers,
  myBusyservers  : QueueObj;      {Queue of idle and busy servers at a point}
  ASK METHOD ObjInit;         {Object initialization method}
  ASK METHOD ObjTerminate;
  ASK METHOD AssignSexAndQType(IN num:REAL) : SexType;
  TELL METHOD EngageServer;
  TELL METHOD Renege;
  ASK METHOD LeaveQ;
  ASK METHOD DoJockeying(IN ser : ANYOBJ);
```

```
  PRIVATE
    ASK METHOD PickQueue() : ServerObj;
END OBJECT;

GeneratorObj =                    {Declaration of Generator object}
OBJECT
  numcustomers : INTEGER;
  TELL METHOD GenCustomers(IN meanInterArrival,simDuration : REAL);
  TELL METHOD GenExtraServer(IN cus : CustomerObj);
END OBJECT;
```

With the development of the above complex queueing model, it can be seen that the object-oriented approach gives a more natural, yet powerful analysis of the systems, thus making simulation programming a natural replication of the real-world system.

3.1.3 Benefits of Object-oriented Approach

The benefits derived from the capabilities of MODSIM as an object-oriented simulation language in developing the above queueing model are tremendous in terms of many aspects. Some chief advantages experienced during developing this model are:

1. The problem-solving approach is more natural with the way humans think and hence, easier and quicker process. This is apparent in this model when it is required to find various objects in the model. The point that the objects in simulation modelling are transparent is a testimonial for using the power of object-orientation for simulation modelling.

2. The inheritance feature of the MODSIM gives programmer a powerful tool for building on the existing data types, thus facilitating already built code to be retained and hence avoiding the problem of defining the new types right from the scratch. In the present model, the queue object inherits the built-in queue object and is developed upon that. Further, the server object subsequently inherits this improved queue object, thus becoming a complex object and this eases the exercise of programming substantially. This has a great bearing on the programmer's productivity.

3. The ability to build complex models is tremendous as is evident from the incremental problem-solving followed in building the queueing model. The data abstraction capability makes this possible, because whenever a feature is to be added to the model, generally the object that is associated only needs changes and so the programmer's concentration can be on that only, whatever may be size of the rest of the code. This has also further advantage in debugging as errors in the modified model can be attributed to the object modified only. This saves debugging time and helps towards faster development of the model.

4. The code maintainability is very good because changes in the implementation of any of the features in the program are *localized* to the region of code of the object that is related to the feature. No fall-out effect is introduced outside the scope of the object.

5. The code is highly transparent, mainly due to the English-like look of MODSIM, thus necessitating no separate documentation of the program.

In addition to these benefits specific to the queueing model, there are many other benefits associated with object-oriented simulation. They are discussed in the following chapter.

3.2 INVENTORY MODELS

3.2.1 MODEL DESCRIPTION

The various inventory models considered are:

Deterministic Models:

1. The Economic Order Quantity (EOQ)
2. The Production Run size Model

Stochastic Models:

3. Variable Demand with *lost-sales* case
4. Variable Demand with *back-order* case

5. Variable Demand and variable lead times with *back-order* case

6. (s,S) Policy

The first two models are determinstic in nature, i.e., the demand and lead times are constant while the other involves variable demand and/or lead time. The last model (s,S) policy somewhat differs from the rest.

3.2.2 Model Development & Implementation

As the model comprises different inventory models which differ from one another in a small way only, this model is well-suited for modular development. In other words, a inventory *library* can be defined for the model consisting of the basic objects that are useful to all the inventory models being developed.

The inventory model, when viewed from the object-oriented analysis, can be visualized with the following objects:

* The Inventory object which is central to the system
* The Supplier object which replicates the supplier of the system and
* The Customer object to represent the demands being generated.

The above objects are declared and implemented in the inventory library. The library consists of two modules, one definition module and one implementation module. Definition module declares the above mentioned objects while the implementation module gives the implementation of various methods of all these objects.

Now, let us consider the fields and methods of each of the objects in turn. The inventory object should have one supplier object type as one of its fields (identifying its supplier), fields for current inventory status, shortages occurred till that point, time of last event, number of inventory holdings, and a random number generator. Methods required are one to process customer demand, one to check for inventory

levels, one to place order, and one to process order arrival. Besides, a method to assign initial inventory is required. Finally, a method to drive the simulation can be incorporated in the inventory object.

The supplier object has fields to note the number of orders placed by the customer, and a random number generator. The methods are to process an order placed by the inventory object and a method to generate lead time.

Finally, the customer object has a field indicating the quantity demanded and a method to update the partial fulfillment of the demand.

The following table gives the list of fields and methods of in the library of the model.

```
SupplierObj =                          {Declaration of Supplier Object}
OBJECT
  orderPlaced : BOOLEAN;
  noOfOrders  : INTEGER;
  ranGen   : RandomObj                 {Random Number Generator}
  ASK METHOD ObjInit;                  {Object initialization}
  ASK METHOD ObjTerminate;             {Object Termination method}
  ASK METHOD LTPattern() : REAL;           {Lead Time Supply method}
  TELL METHOD ProcessOrder(IN Inv : ANYOBJ);
END OBJECT;

InventObj =                            {Declaration of Inventory object}
OBJECT
  supplier   : SupplierObj;
  inventory,
  shortage   : INTEGER;
  lastChangeTime : REAL;
  holdingCost : REAL;
  ranNumGen : RandomObj;               {Random Number Generator}
  TELL METHOD GenDemand(IN sup : ANYOBJ);
  ASK METHOD AssignInv0;                 {Assign initial inventory}
  ASK METHOD CheckInvLevel;
  ASK METHOD OrderArrival;
  ASK METHOD PlaceOrder;
  TELL METHOD ProcessDemand(IN dem : ANYOBJ);
  ASK METHOD DemandPattern() : INTEGER;
END OBJECT;

CustomerObj =                          {Declaration of Customer object}
OBJECT
  demand : INTEGER;
  ASK METHOD AssignDemand(IN num : INTEGER);
```

```
END OBJECT;
```

Once the library is created, it is a simple task to develop the various models using the objects defined in the library. To generalize the things, every *main* module imports the objects defined in the library, override the required methods (in particular, the demand and lead time pattern), and runs the simulation through the main routine. (No value is placed in the library's methods that give demand and lead time values hence these methods are to be overridden necessarily in every main module.) Results are printed through a procedure after simulation run is over.

Consider the simple EOQ model. The model is directly built from the library, with minor modifications to the demand and lead time methods. The module imports the objects from the library and modify as shown below:

```
ThisInventObj =
OBJECT(InventObj)
OVERRIDE
  ASK METHOD DemandPattern() : INTEGER;
END OBJECT

ThisInventObj =
OBJECT(SupplierObj)
OVERRIDE
  ASK METHOD LTPattern(): REAL;
END   OBJECT
```

Hence the objects are inherited to build objects suitable to the model, thus reusing the rest of the code defined and implemented in the library. In this way, the code required for the model in its main module is negligible compared to what it would have been in case of full development of the model in a single module.

The production run size model needs some more minor modifications than the simple EOQ. It requires the production rate to be inputted and uses it as the order quantity. Otherwise, this model also makes full use of the inventory library.

Among the stochastic models, the variable demand with the last-sales case involves only overriding the demand pattern method appropriately.

The other models are of back-order case, i.e., the shortage demand is met once the order placed is received. This requires a modification to the inventory object defined in the library created earlier and appropriate overriding of its methods. This modification is mainly in terms of creating one queue object as the field in the inventory object to keep unsatisfied customer demands. So, the new library imports the inventory object defined in the older library, inherits it, and adds the required new field to build the new inventory object. Since this modification is required by all other main modules to be constructed, one modified inventory library may be appropriate. Hence, one more library is created with the inventory object defined as below:

```
QInventObj =
OBJECT(InventObj)
demandQ   : QueueObj;
OVERRIDE
  ASK METHOD AssignInv0;
  TELL METHOD ProcessDemand(IN sup : ANYOBJ);
  ASK METHOD OrderArrival;
END OBJECT;
```

All the main modules that involve back-ordering, thus, import the inventory object of the revised library. The rest of the implementation is imported from older library. Consider the variable demand case with back-ordering. This main module imports the inventory object defined in the new library, and the supplier and customer objects from the older library, and overrides the demand and lead time methods as ever. In this way, this module makes use of both inventory libraries to build powerful model with less code.

The remaining model is also built on similar lines. It is a model with variable demand and variable lead times, which is achieved by

just overriding the methods of demand and lead time respectively.

The (s,S) policy model retains the inventory object of the older library, but requires major additions of fields and methods besides overriding some methods. This is because the model involves *periodic review*, and this is achieved by MODSIM in an elegant way. It requires a field to store the period of review. It also requires a TELL method that schedules the review of inventory level at the end of each review period. The inventory object, inherited from the older library, is modified as below:

```
ThisInventObj =
OBJECT(InventObj)
policyQuantity   : INTEGER;
reviewTime       : REAL;
TELL METHOD Review;
OVERRIDE
  ASK METHOD AssignInv0;
  ASK METHOD CheckInvLevel;
  ASK METHOD DemandPattern() : INTEGER;
  TELL METHOD ProcessDemand(IN sup : ANYOBJ);
  ASK METHOD OrderArrival;
END OBJECT;
```

The field "policyQuantity" is the size of the order which is determined by the TELL method *review*. The model imports the supplier object and the customer object as it is from the library and their code remains valid to build the model.

3.2.3 Benefits of Object-oriented Approach

The capabilities of MODSIM as an object-oriented simulation language is exploited to a great extent in building the set of inventory models described above. These are explained below:

1. Modular Development:

The complete program is built in modules by creating libraries of inventory and making use of these libraries, main modules are created for each model considered. In this way, the same code (in the libraries) serves all the models, hence minimizing the coding task of the programmer.

2. Encapsulation:

The objects are defined in the library and their methods are implemented there only. Hence a programmer who wants to make use of the library for building appropriate models need to know only what various methods of the objects do. He doesn't have to worry about the actual implementation details of these methods. This saves considerable amount of time the programmer spends on understanding the existing code before extending the model for more features. All this is possible as the data and methods are *encapsulated* in the object data type.

3. Inheritance:

Inheritance again proved to be a powerful capability in that the objects defined in the libraries are inherited and are modified further to suit the specific application, while retaining the implementation details of the object methods wherever necessary.

4. Polymorphism:

Though the inherited objects are modified to suit the specific application, the derived types are still consistent with the base types and so makes it possible to utilize the code along with the other objects imported from the library.

5. Ease of building the models:

Once the basic object types are declared in the library, it is easier to build upon any number of models, as long as the basic objects of the system don't change.

6. Length of the code

This is a major benefit experienced during building this simulation model. The entire code for the six models developed including the libraries come to less than seven hundred and fifty lines while in conventional programming this would be atleast four fold! This is made possible by the reusability

of the code and inheritance and other properties of the MODSIM. Further, much of this code is replication in the main modules which are easily carried out with sophisticated editors available presently.

CHAPTER 4
CONCLUSIONS & RECOMMENDATIONS FOR FURTHER WORK

4.1 CONCLUSIONS

The benefits that can be derived from carrying out the simulation modelling with the object-oriented simulation language, MODSIM are demonstrated in the present work. The chief advantages that are experienced by modelling with MODSIM are explained below:

1. *Natural Approach to Problem-Solving*

The use of object-orientation in simulation modelling provides a *natural* approach to problem solving. In other words, it is compatible with the way humans think during problem-solving. This has potential advantages in terms of the validity of the model developed, since the data manipulation is clear. This is made possible by the basic philosophy of OOP to concentrate on the system rather than the algorithms for solving the problem.

2. *Ability to Build Complex Models with Ease*

It is common that the real-world systems that are subjected to simulation analysis are very complex in nature and so require high programming skills on the part of the modeller. However, MODSIM, with its object-oriented capabilities like data abstraction, inheritance, and polymorphism, makes it considerably easier to build these real-world models.

3. *Improved Programmer's Productivity*

The two key properties of object-oriented programming, inheritance and polymorphism of the code help in improving the programmer's productivity significantly. Also, the debugging time can be cut drastically due to properties like inheritance.

4. *Generalization of code*

The code developed for data types, especially objects, that are common to many simulation models can be stored in user-defined libraries and can be

imported by some other applications, thus avoiding the exercise of defining them from the scratch. This is called *reusability* of the code which is a powerful feature of OOP.

5. *Faster Development of the Project*

Since the models can be built in modules, MODSIM allows multiple-programmers to work on the same project, thus expediting the development process. This is of significance when the projects are time-constrained.

6. *Length of Code*

The length of code generated for a specific model in MODSIM is generally many times lesser than that generated in a procedural-language. This is again made possible by the features of object-orientation of MODSIM.

7. *Readability of the Code*

MODSIM has a English-like look which makes the code easily readable. Also, no separate documentation is required as the code is transparent. This is very comfortable from a new-user point of view.

In this way, MODSIM, as an object-oriented programming language with special capabilities for discrete event simulation, surpasses the abilities of the traditional languages. Also, programmers can learn the language fast due to its Pascal-like look.

However, the simulation with object-orientation is not completely cake-walk. Some major limitations are:

1. For the programmers who are accustomed to the procedural paradigm, thinking in object-oriented analysis way is no easy matter. Hence, the complete benefits of MODSIM can be explored only after the programmer obtains sufficient skills in the object-oriented analysis.

2. The benefits of object-orientation can be fully utilized only when it is applied to large projects. For small and medium level projects, the gains may not be substantial compared to the procedural languages.

3. As MODSIM supports *discrete event* simulation only, systems that are to be modelled for *continuous* simulation can not use MODSIM.

4. The languages that support the object-orientation are very costly compared to the traditional languages.

5. The languages that support the object-orientation are generally computer memory-expensive.

The following table compares various special-purpose simulation languages [12] *vis-a-vis* the MODSIM. The rating of the MODSIM has been done based on the experience in carrying out the present work.

Comparison of various Simulation languages:

Criterion	Language GASP IV	SIMSCRIPT	GPSS	MODSIM
1. Types of systems oriented toward	General	General	Queueing	General
2. Event-oriented approach	Yes	Yes	No	Yes
3. Process-oriented Approach	No	Yes	Yes	Yes
4. Mechanisms for getting random variables easily	Yes	Yes	Yes	Yes
5. Natural framework for simulation modelling	Fair	Good	Very Good	Out-standing
6. Ease of learning	Very Good	Fair	Good	Excellent
7. Language flexibility	Excellent	Excellent	Fair	Excellent
8. Language programming power	Fair	Excellent	Excellent	Out-standing
9. Communication ability	Fair	Very Good	Excellent	Out-standing
10. Combined discrete-continuous simulation	Yes	Yes	No	No
11. Relative cost of purchase	Low	High	High	Very High

4.2 RECOMMENDATIONS FOR FURTHER WORK

Certainly, developing two simulation models won't lead to generalization of things. The testimony for any simulation language can be the usefulness of the models developed. Hence, the power and capabilities of MODSIM can only be realized when extensive use of the language is made for simulation purposes. The further work can ,thus, be in the form of using MODSIM to build models of diverse systems like those in manufacturing, finance, marketing, economics. In particular, simulation of flexible manufacturing systems (FMS) is felt to be well-suited for MODSIM due to the intense interaction between the objects in such systems. Also, the complexities of economic models make them appropriate for MODSIM.

Further research on MODSIM can be done in terms of comparing it as a simulation language with other object-oriented languages, especially C++. In view of the emergence of C++ as the language of the software development, more programmers might be familiar with it. Further, the capabilities of C++ like operator overloading, declaring fields and methods as *protected* (besides private and public), and concepts like *Friends* which are absent in MODSIM, may make C++ more powerful than MODSIM for simulation modelling. Therefore, a comparative study of MODSIM with C++ may prove worthwhile.

Further work can also be done in other aspects of MODSIM which are not given attention in this work, like the statistical analysis of simulation output data, debugging facilities, etc.

REFERENCES

[1] Bezivin, J., "Some Experiments in Object-Oriented Simulation," OOPSLA Proceedings, 1987, pp.394-405

[2] Bratley, P., B.L.Fox and L.E.Schrage, "A Guide to Simulation," Springer-Verlag, New York, 1983

[3] CACI Products Company, "MODSIM II: The Language for Object-Oriented Programming: Reference Manual," California, 1992

[4] CACI Products Company, "MODSIM II: The Language for Object-Oriented Programming: Tutorial," California, 1992

[5] CACI Products Company, "MODSIM II: The Language for Object-Oriented Programming: User's Manual," California, 1992

[6] Computers Today, "Software Made Easy," November 1992, Vol 8, No. 93, 32-39

[7] Corey, P.D. and J.R.Clymer, "Discrete event simulation of object movement and interactions," *Simulation*, March 1991, Vol. 56, No. 3: 167-174

[8] Dahl, O.J., and K.Nygaard, "SIMULA - An Algol-based Simulation Language," *Communications of the ACM,* September 1966, Vol 9, No.1, 671-678

[9] Eldredge, D.L., J.D.McGregor and M.K.Summers, "Applying the object-oriented paradigm to discrete event simulations using the C++ language," *Simulation*, February 1990, Vol. 55, No. 2: 83-91

[10] Fishman, G.S., "Concepts and Methods in Discrete event Digital Simulation," John Wiley & Sons, New York, 1973

[11] Gordon, G., "System Simulation," Prentice-Hall, Englewood Cliffs, N.J., 1969

[12] Law, A.M. and W.David Kelton, "Simulation Modeling and Analysis," McGraw-Hill Book Company, New York, 1982

[13] Payne, J.A., "Introduction to Simulation: Programming Techniques and Methods of Analysis," McGraw-Hill Book Company, New York, 1988

[14] Ravindran, A., D.T.Phillips and J.J.Solberg, "Operations Research: Principles and Practice," John Wiley & Sons, New York, 1987

[15] Russell, E.C., "Building Simulation Models with SIMSCRIPT II.5," C.A.C.I., Los Angeles, 1983

[16] Smith, N.E., "Object-Oriented Programming with Turbo C++," BPB Publications, New Delhi, 1992

[17] Starr, M.K., and D.W.Miller, "Inventory Control: Theory and Practice," Prentice-Hall, Englewood Cliffs, N.J., 1962

[18] Wiener, R.S. and Pinson, L.J., "An Introduction to Object-Oriented Programming and C++," Addison-Wesley Publishing Company, Massachusetts, 1988

[19] Zeigler, B.P., "Hierarchical, Modular Discrete-Event Modelling in an Object-oriented Environment," *Simulation*, May 1986, Vol. 49, No. 5: 219-230